COMP 3520 - ASSIGNMENT 2

DESIGN DOCUMENT

THE HYPOTHETICAL OPERATING SYSTEM TESTBED

(HOST) DISPATCHER SHELL

Name: Sebastian Kim

TABLE OF CONTENTS

A. Memory Allocation Algorithm 2

Justification of Choice 3

B. Structure of Dispatcher 3

1. Queue Structure 3

2. Memory Structure 4

3. Resource Structure 5

C. Overall Structure of Program 5

Interface 5

1. HOST.C – Hypothetical Operating System Testbed 5

2. PCB.C – Process Control Block 6

3. MAB.C – Memory Allocation Block 6

4. RSRC.C – Resources 6

Justification of overall structure 6

D. Further Discussion 7

1. Dispatcher 7

Shortcommings and Possible Improvements 7

2. Memory Allocation 7

Shortcommings and Possible Improvements 7

3. Resource Allocation 8

Shortcommings and Possible Improvements 8

References 9

A. MEMORY ALLOCATION ALGORITHM

There are two ways of which memory can be partitioned: fixed and dynamic partitioning. With fixed partitioning, all memory blocks have the same size. Due to this reason, fixed partitioning can cause inefficiency of memory space (internal fragmentation) and overlaying issues. On the contrary, dynamic partitioning method could lessen both problems by dividing memory blocks into variable sizes according to the requirement of processes.

The implementation of dynamic partitioning can be done with various allocation algorithms including First-Fit, Next-Fit, Best-Fit, Worst-Fit and Buddy System. Each allocation method has different characteristics having distinctive advantages and disadvantages.

- **First-Fit** allocates processes at the very first available memory blocks. It allows simple and fast memory allocation, but it will slow down as the portion of memory used grows because most of processes located in the front end of the memory. Also, there are potential dangers of external fragmentation problems, which decrease utilisation of the memory space.
- **Next-Fit** behaves similar to First-Fit, but starts scanning memory from the place of previous allocation, not from the beginning of memory. This algorithm allocates memory faster than the First-Fit, but the overall performance is inferior to First-Fit (Bays, 1977). Next-Fit also has external fragmentation issues.
- **Best-Fit** allocates memory at the smallest possible block for processes, thus it can minimise total size of the unusable blocks caused by external fragmentation. However, searching the best-fitted place through all memories every time of allocation can be both time consuming and wastes of CPU. It is therefore known as the worst performer among them.
- **Worst-Fit** selects largest possible block fits the process. This is the opposite idea to the Best-Fit algorithm. Best-Fit leaves the external fragmentations but minimise the size of total unusable blocks, whereas Worst-Fit prevents external fragmentations by splitting the largest size of memory blocks (because the remaining fragments are still big enough to use). Like Best-Fit, Worst-Fit algorithm wastes CPU time for searching the largest block in the whole memory space.
- **Buddy system** is based on more complex implementation with binary tree data structure(?). This ensures faster scanning of memory, but it has the fragmentation

problems as other algorithms have. According to Page (1986), the typical implementation of Buddy System has just 65 percent of storage utilisation level while First / Best-Fit algorithm showed average 80 percent of utilization with the same input.

JUSTIFICATION OF CHOICE

I chose First-Fit memory allocation algorithm in this assignment as the follow reasons:

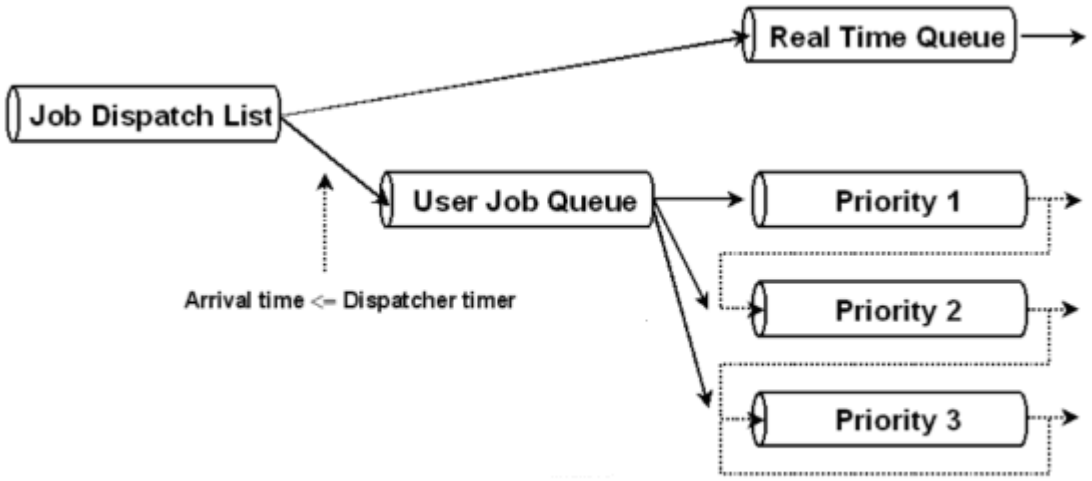
- There are general agreements based on previous experiments that First-Fit showed acceptable performance and good memory utilization.
- First-Fit can be implemented with just few lines of code and it is easy to understand.
- Although, Buddy System can enable fastest memory allocation, it is inferior to First or Best-Fit in terms of memory utilization. Moreover, there are plenty of ways to implement Buddy Systems, but even the basic one is too complicated. Therefore, it is not appropriate for the purpose of this assignment.

B. STRUCTURE OF DISPATCHER

1. QUEUE STRUCTURE

There are total 6 queues used to implement multi-level feedback queues. Each queue is implemented with a basic linked list of process blocks. Job dispatch list (input queue) will be filled with an input file (up to 1000 jobs), and will feed the jobs into either real-time queue or user job queue according to its priority. User job queue will dispatch the jobs into appropriate priority queues if there are enough memory space and resources available. The real-time jobs will never be pre-empted by other processes.

Each job can have 4 different levels of priorities, and priority 0 is regarded as the highest priority (real-time jobs), and all real-time jobs will be executed in First-Come-First-Serve manner. Lower level priority jobs (from 1 to 3) in the user job queue will be fed into appropriate priority queues. Every time quantum passed, running process can be pre-empted if there is any other processes waiting. If a job is suspended by other processes, its priority will be decreased by 1 level and dispatched at the end of one level lower priority queue. Figure 1 below illustrates this logic flow.



< Figure1. Dispatcher Logic Flow >

2. MEMORY STRUCTURE

Memory structure has been implemented with doubly linked list for the convenience of merging and splitting operations of memory blocks. Each memory block is linked with its next and previous memory blocks and each memory block has the information of its offset, size, allocation flag, etc. Memory will be allocated for a process during the life-time of the process (from start to finish time).

The system has total 1024MB of memory, and the first 64MB is reserved for real-time jobs. This reserved space is separated from the user job space (960MB) because real-time space should not be merged with user job space. There is no merging and splitting operations needed in real-time space because only one job can be executed at any given time. Thus all 64MB of space will be allocated to whatever the requirement of process is. If a real-time job requires more than 64MB of memory space, it will be regarded as an “invalid input” and will be deleted from the input queue.

On the other hand, another 960MB of memory space will be used by user jobs, and the memory block will be allocated dynamically with First-Fit algorithm. Every process will be allocated into exactly one memory block. If the size of selected block is bigger than the requirement of the process, the block will be split into two blocks; left part will be allocated, and right one will be freed. When a process terminated, the allocated memory will be freed and possibly merged with adjacent free blocks.

3. RESOURCE STRUCTURE

There are 4 different types of I/O resources: 2 Printers, 1 Scanner, 1 Modem and 2 CD-Drives. Available system resources implemented as a global variable, and will be allocated from the start to finish time of the process (same as memory).

C. OVERALL STRUCTURE OF PROGRAM

The program consists of 4 major parts of implementations:

- hostd.c: implementation of job dispatcher and overall logic flows
- pcb.c: implementation of process control block
- mab.c: implementation of memory allocation block
- rsrc.c: implementation of I/O resources

Interface of each component will be described below (unimportant functions may be omitted).

INTERFACE

1. HOST.C – HYPOTHETICAL OPERATING SYSTEM TESTBED

- readFile(): Read input file and fill the job dispatch list (input queue) with the inputs.
- validateJob(): Validate a job, according to whether it requires excessive resources or not. Invalid jobs will be deleted before it goes into ready queues.
- deqInputQ(): Dispatch a job from the input queue and put it into either real-time queue or user job queue.
- deqUserJobQ (): Dispatch a job from the user job queue and put it into appropriate priority queues.
- main(): Main function contains all the logic flows.
    - nanosleep() function is used to guarantee that the process is not executed overtime. When I implemented this with normal sleep() function, there was about 20% of errors randomly occurred because sleep() function can be easily affected by other processes or the system conditions, so the time in “hostd” and “process” can be mismatched. Using nanosleep() function helped to resolve this problem, but there are still very few over-running of process can be occurred.

2. PCB.C – PROCESS CONTROL BLOCK

- startPcb(): Start a process(execute "process" in current directory). If the process is already started, but suspended, restart the process by sending SIGCONT signal.
- suspendPcb(): Suspend a process by sending SIGTSTP signal.
- terminatePcb(): Terminate a process by sending SIGINT signal.
- createnullPcb(): Create a process with initialisation.
- enqPcb(): Enqueue a process at the end of the queue.
- deqPcb(): Dequeue a process from the head of the queue.

3. MAB.C – MEMORY ALLOCATION BLOCK

- memInitialise(): Initialise a null memory block.
- memChk(): Traverse the memory and return the first available block for the process.
- memMerge(): Merge adjacent memory blocks if the blocks are not allocated. Return the merged block of free memory.
- memFree(): Free a memory block, and merge it if possible
- memSplit(): Split a memory block. Left partition will be allocated and right will be freed. Merge right block with the next one if the next one is free.
- memAlloc(): Allocate a process into a free space. If the block size is bigger than the process requirement, splits and allocates the process into a fitted size.

4. RSRC.C – RESOURCES

- rsrcAlloc() : Allocate the resources to the process when it is started.
- rsrcChk(): Check there is enough resources available.
- rsrcFree(): Free the resources which a process used when it is terminated.

JUSTIFICATION OF OVERALL STRUCTURE

Files are divided by logical classes of implementations which resemble object oriented design. All the functions and logics are highly modulated to improve readability and extensibility.

In the assignment description, job validation is performed on user job queue, and just assumes all real-time job inputs are valid. However, I implemented this on input queue because real-time job inputs also need to be checked for more consistent outputs. Thus we can assume all the jobs in both user job queue and real-time queue are valid because invalid jobs are already deleted on its arrival time.

D. FURTHER DISCUSSION

1. DISPATCHER

Multi-level feedback queues are widely used in modern Operating Systems (OS). This maximises the CPU utilization by scheduling jobs efficiently, as well as ensures better user experiences by putting more preferences to short jobs and I/O bounded processes. However, the implementation of the dispatcher in real operating system is much more complex than the one used in this assignment because there are various situations need to be handled. For instance Windows NT has 32 levels of priority levels and Linux has priority level from 0 to 140. Furthermore, Windows Vista has implemented a special scheduler for multimedia classes for better playback experience of multimedia files (Russinovich, 2007).

Real OS can also have different time quantum depending on different situations whereas our implementation has fixed 1 second of time quantum. One second of time quantum is actually too long for real situation and it is usually has a range from few milliseconds to few hundred milliseconds.

SHORTCOMMINGS AND POSSIBLE IMPROVEMENTS

Our program does not use thread concept which can help to decrease overload of pre-empting / switching of processes. Also, this only supports single core processor whereas most computers nowadays have more than dual core CPUs.

2. MEMORY ALLOCATION

Similar to our program, general operating systems tend to use dynamic memory allocation method because of the efficiency of memory space utilisation. This is however implemented with more complex algorithms (e.g. Buddy System) to improve the performance as well as lessen the problem of allocating algorithms. For example, memory compaction will be essential to resolve fragmentation issues, and chunking can be used for handling small allocations.

SHORTCOMMINGS AND POSSIBLE IMPROVEMENTS

Our program cannot run a process which requires more than 64MB and 960MB of memory for real-time jobs and user level jobs respectively. It just deletes the process if the process

requires excessive memory space, thus it is impossible to handle heavy programs. If we implement virtual memory concept with paging, those problem can be partially solved.

Also, by using some more complex algorithms (e.g. Buddy System, Compaction, Chunking, etc.) the performance of memory allocation can be improved.

3. RESOURCE ALLOCATION

In this assignment, the resource allocation has been implemented in very basic level because we assumed all the resources allocated at the starting time of the process and will be freed when the process is terminated. However, in reality, this is impractical because processes could request more resources while it is running, and some of limited resources can to be shared with other processes at the same time. For example, even though a web browser is using the modem, other processes (e.g. Messenger, E-mail client) might also need to be able to access the modem. In real situation, therefore, deadlock issues can arise and complicated methods (synchronisation) will be needed to solve this problem.

SHORTCOMMINGS AND POSSIBLE IMPROVEMENTS

In my opinion, our implementation of resource allocation cannot represent the real situations. It would be better practice if we implemented this more in detail including synchronisations.

REFERENCES

CARTER BAYS. 1977. A comparison of next-fit, first-fit, and best-fit. *Magazine Communications of the ACM*, 22

MARK RUSSINOVICH. 2007. Inside the Windows Vista Kernel: Part 1. *TechNet Magazine*. Available: http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx

PAGE I.P, HAGINS J. 1986. Improving the Performance of Buddy Systems. *Computers, IEEE Transactions on*. C-35, 5, 441-447. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1676786&isnumber=35256